Space-efficient Planar Acyclicity Constraints - A Declarative Pearl

Taus Brock-Nannestad

Space-efficient Planar Acyclicity Constraints
A Declarative Pearl

Taus Brock-Nannestad

Inria & LIX/École polytechnique
taus.brock-nannestad@inria.fr

Abstract. Many constraints on graphs, e.g. the existence of a simple path between two vertices, or the connectedness of the subgraph induced by some selection of vertices, can be straightforwardly represented by means of a suitable *acyclicity* constraint. One method for encoding such a constraint in terms of simple, local constraints uses a 3-valued variable for each edge, and an $(N + 1)$-valued variable for each vertex, where N is the number of vertices in the entire graph. For graphs with many vertices, this can be somewhat inefficient in terms of space usage.
In this paper, we show how to refine this encoding into one that uses only a single bit of information, i.e. a 2-valued variable, per vertex, assuming the graph in question is planar. We furthermore show how this same constraint can be used to encode *connectedness* constraints, and a variety of other graph-related constraints.

1 Introduction

In this paper, we aim to present an "encoding pearl" that shows how to encode various graph constraints in terms of an acyclicity constraint, and also how to decompose such an acyclicity constraint into a space-efficient (in terms of the combined sizes of the variable domains) collection of low-level constraints.

Our acyclicity constraint can be seen as a refinement of an intuitive, obviously correct, but space-inefficient constraint, which, to the best of our knowledge, is due to Tamura [5], although it is an obvious enough encoding that it may simply be *folklore*. To the best of our knowledge, the space-optimised constraint we present here is novel.

We will present our constraints both using a high-level, prose description, but also in terms of the more explicit language of *finite linear integer constraints*. We choose this as the target for our encoding because of its flexibility — we will freely make use of the fact it straightforwardly permits the encoding of e.g. conditionals and reified constraints.

In many cases, using a specialised acyclicity constraint, such as the one presented by Gebser *et al.* [1], will be more efficient for finding solutions to constraint satisfaction problems, as it can use domain-specific knowledge to propagate constraints in a way that a naïve encoding may not be able to.

On the other hand, there are often large gains to be had from encoding a problem using high-level constraints into e.g. a boolean satisfiability (SAT) problem, as seen in the *Sugar* CSP solver [6].

Given the above considerations, we make no claims about the *real-world* efficiency of the solution we present in this paper, and rather present it as a neat theoretical *curiosity*.

The remainder of the paper is structured as follows. In Section 2, we show how acyclicity plays a crucial rôle in encoding path constraints. In Section 3, we present a straightforward but space-inefficient encoding of such a constraint, in a subset of planar graphs which we call *grid graphs*, followed by a few improvements in Section 4. In Section 5, we present our optimised encoding, and we prove its correctness in section 6. In Section 7, we show how to extend the results concerning grid graphs to general planar graphs without a loss in efficiency. Section 8, we consider other kinds of graph constraints, and show how these can also be encoded using our acyclicity constraint. Finally, in Section 9, we conclude and discuss future work.

2 Making use of acyclicity

In this section, we will show exactly how an acyclicity constraint can be used to correctly enforce certain constraints on graphs. First, let us assume we are given some fixed graph G with two distinguished vertices s and t, and that we wish to select some subset of the edges so that they make up a single simple path from s and t. We will associate a variable to each edge of the graph, and say this variable has the value 1 if the edge is selected, and 0 otherwise. To ensure that the path is simple, we can add constraints that restrict the number of selected edges meeting a vertex as follows:

- Around s and t, we require that there is exactly one selected edge.
- For any other vertex v, we require that the number of selected edges around v is either 0 or 2.

These constraints already eliminate many incorrect selections, but unfortunately not all of them. The problem is that the above constraints ensure that the solution must be path-like around each vertex, but this is not enough:

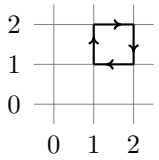
Any part of a cycle is *also* path-like around each vertex in the cycle, hence we may get spurious cycles with just the above constraints. At this point it should hopefully be clear that if we find some way of preventing cycles from appearing in our assignment of edges, we may ensure that the solution consists of only a single, simple path.

Before we present a way of encoding such an acyclicity constraint, we will briefly remark on two simplifying assumptions we can make in this setting. First of all, we may assume that the graph in question is simple, i.e. there is at most one edge between any two vertices, and no edges from a vertex to itself. This can

be justified by noting that if we have selected two edges that have the same start and end vertices, then we have already created a cycle. Similarly, any edge that is not part of any cycle of the graph can also be disregarded for the purposes of enforcing acyclicity. Thus, in the remainder of this paper, we will assume that the underlying graph is simple and bridgeless.

3 A basic encoding of the acyclicity constraint

First, we'll describe an inefficient, but hopefully intuitive encoding of an acyclicity constraint. To simplify the presentation, we'll only present the results in this section for a very simple and well-behaved graph. The graph we will consider has a vertex for each point $(i, j) \in \mathbb{Z} \times \mathbb{Z}$, and edges between any two vertices that are a unit distance apart. For the variables representing these paths, we will use $h_{i,j}$ for the horizontal edge extending rightwards from the vertex at (i, j), and $v_{i,j}$ for the edge extending vertically at this vertex. Furthermore, we will in certain situations rely on these edges having values that indicate not just whether they are selected, but also marking them with a specific direction. In this case, we will assume that the variables $h_{i,j}$ and $v_{i,j}$ may take on values from the set $\{-1, 0, 1\}$. A negative value indicates that the direction of the edge is to the left or down, and a positive value indicates that the direction of the edge is to the right or up. Thus,



is represented by $\quad h_{1,1} = -1,\ v_{1,1} = 1,\ h_{1,2} = 1,\quad v_{2,1} = -1$

To avoid having to consider truly infinite graphs, we will assume that all variables that are indexed by positions (i, j) take on the default value 0 on all but finitely many points. This essentially restricts us to working within a finite subgraph of the grid graph. The benefit of doing this, as opposed to working with finite graphs to begin with, is that this presentation allows for a uniform presentation of the constraints, and in particular avoids nasty boundary conditions that might otherwise arise.

Essentially, the constraints we will add have the following effect:

- First, all paths are forced to be *simple* — paths may not touch or cross each other:
$$\forall i, j. \quad |h_{i,j}| + |v_{i,j}| + |h_{i-1,j}| + |v_{i,j-1}| \in \{0, 1, 2\}.$$
- Secondly, all paths are forced to be *directed*. If a vertex has two edges connected (which is the maximum as per the previous constraint), then one edge must be pointing *towards* the vertex, and the other must point *away* from the vertex. This can be encoded as follows:
$$\forall i, j. \quad |h_{i,j} - h_{i-1,j} + v_{i,j} - v_{i,j-1}| \leqslant 1$$

This constraint may seem a bit unintuitive, but it is a straightforward matter to check that this (along with the preceding constraint) forces paths to be directed.

Next, we associate a variable $u_{i,j}$ to each vertex (i, j). This variable will take on values from the set $\{0, \ldots, N\}$ where N is the number of vertices of the subgraph we are considering. These variables are constrained as follows:

– Along any directed edge, the value of the source vertex must be strictly greater than the value of the target vertex:

$$\begin{aligned}
&\forall i, j. \quad (h_{i,j} > 0) \supset (u_{i,j} > u_{i+1,j}) \\
&\forall i, j. \quad (h_{i,j} < 0) \supset (u_{i,j} < u_{i+1,j}) \\
&\forall i, j. \quad (v_{i,j} > 0) \supset (u_{i,j} > u_{i,j+1}) \\
&\forall i, j. \quad (v_{i,j} < 0) \supset (u_{i,j} < u_{i,j+1})
\end{aligned}$$

The effect of these constraints is the following: first, all cycles must become *directed* cycles, as enforced by the first constraint. Second, the vertices appearing in a directed path must have values that are strictly decreasing along this path. Having both of these constraints thus precludes any cycles from appearing in the graph. Of course, this constraint might be too strict, for instance by excluding paths that should be allowed, but it's easy to see that any directed path can have strictly decreasing values assigned to it, since the domain of these values is larger than the number of vertices in the graph.

The main problem with this encoding is the size of the domain of the values. We require that each vertex (i, j) has a unique variable $u_{i,j}$ associated to it, and that these variables may take on values from the domain $\{0, \ldots, N\}$ where N is the total number of vertices in the graph. Thus, the more vertices our graph has, the greater this domain must be, leading to a quadratic growth in terms of space.

4 Potential refinements

In this section, we will explore a few possible refinements that unfortunately do not quite improve matters. Although it is not customary to present approaches that do not work, we believe that in this case it gives a useful glimpse into the genesis of the encoding that will be presented in the next section.

The first and most obvious way of reducing the domains of the vertex variables would be to just reduce the domain. Unfortunately, this will in general also exclude some paths, which is not always desirable. The main problem with this approach is that the values along a path must still be strictly decreasing, hence whatever bounds we put on the values will also be a bound on the lengths of the paths it is possible to represent in the graph.

Of course, it is not necessary to strictly decrease the value along *every* edge in the graph. We could instead require that the values are non-increasing everywhere, as long as we have some guarantee that it will be strictly decreasing along at least one edge of any given cycle.

One way of enforcing this would be to pick a subset of the edges in advance, and tailor the constraints to be enforce a strict decrease along these edges, and allow any kind of decrease everywhere else.

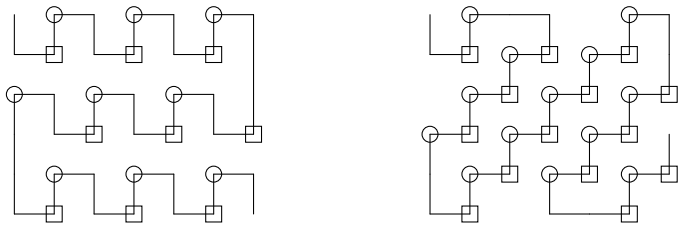
How should one choose such a subset? An easy way to do so is to find a spanning tree of the graph, and then select all the edges that are *not* in this spanning tree to be the strictly decreasing ones. This works because any cycle must contain at least one of these edges, since the tree by definition cannot contain any cycles.

How much of an improvement is this? In a graph with N vertices, the spanning tree will contain $N - 1$ edges, and hence the number of strictly decreasing edges will be $E - (N - 1)$ where E is the number of edges in the graph. In the worst case, there exists a path that traverses all of these edges, and hence the domain of values associated to the vertices must have at least as many elements as the length of this path. In the case of the grid graph, we can define a spanning tree by taking all of the vertical edges, and a single horizontal line of edges. This leaves roughly half the edges of the graph as strictly decreasing.

Instead of choosing the strictly decreasing edges in advance, we might also select them in a more dynamic fashion. The basic idea is the following: if we can identify some feature (or set of features) that is guaranteed to be part of every cycle, we can use this to add strictly decreasing edges only at the points where these features occur. For instance, every cycle must contain at least one top-left corner, hence we could choose to make the vertical edge of each such corner strictly decreasing.

This is again an improvement on the previous situation, as the number of such corners in any path can be bounded in advance. Note that any top-left corner cannot have a similar corner immediately below or to the right, hence in a grid with N vertices, $N/3$ is certainly an upper bound on the number of top-left corners. Unfortunately, we can still construct paths that contain many top-left corners, hence this is not an asymptotic improvement on the efficiency of the encoding.

Here are two examples of paths that have many top-left corners, which we have marked using circles:



If we consider these and other examples, it quickly becomes apparent that any path that contains many top-left corners must also contain many bottom-right corners, which are marked with squares above. Moreover, if we follow the paths given by these examples, we find that we *alternate* between top-left and bottom-right corners.

5 An optimised encoding

As the refinements in the previous section hinted, there seems to be a relationship between the number of top-left corners and bottom-right corners.

First, we will change our encoding slightly. Instead of requiring a set of values to be (strictly) decreasing along the edges, we will instead keep just a single bit of information at each vertex. Our goal will be to give conditions that impose either equality or disequality constraints between vertices that are connected by an edge, in such a way that any cycle must contain an *odd* number of disequality constraints. If we can ensure this behaviour, then we will have succeeded in disallowing all cycles. Intuitively, we can imagine walking along the cycles with a single bit, following the directed edges, and flipping the value of our bit whenever we encounter a certain configuration of edges. As long as we make an odd number of flips, we are guaranteed to end up with the opposite parity when we return to the starting position. Naturally, paths with be unaffected by these constraints.

To encode these constraints, we first assign a binary-valued variable $p_{i,j}$ to each vertex (i, j). We next add the following constraints:

- For all edges directed downwards or to the left, we simply enforce equality between the values at each end:

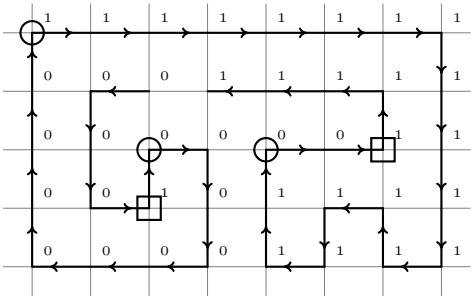
$$\forall i, j. \quad (h_{i,j} < 0) \supset (p_{i+1,j} = p_{i,j})$$
$$\forall i, j. \quad (v_{i,j} < 0) \supset (p_{i,j+1} = p_{i,j})$$

- For edges directed upwards or to the right, we similarly constrain the values at each end, but choose whether to enforce equality or disequality depending on whether the edges in question form an "up-then-right" or "right-then-up" corner:

$$\forall i, j. \quad (h_{i-1,j} > 0) \supset (p_{i-1,j} = p_{i,j} \oplus (v_{i,j} > 0))$$
$$\forall i, j. \quad (v_{i,j-1} > 0) \supset (p_{i,j-1} = p_{i,j} \oplus (h_{i,j} > 0))$$

  Here, $\oplus$ is the "exclusive or" operation.

Note that these four constraints cover all the possibilities for which the value of $h_{i,j}$ and $v_{i,j}$ is non-zero. Moreover, the four constraints are also *disjoint*, i.e. for any given edge, at most one of the above constraints apply. Here is an example that shows how the above constraints force the assignment of the $p_{i,j}$ values for a particular directed path (where we have arbitrarily set the value of p at the beginning of the path to 0):



The circles and squares indicate the vertices at which the flips take place. Observe that if we were to close the cycle by adding an extra edge going left, there would be no way of reconciliating the values of p.

All that remains now is to show that closed simple cycles must always contain an odd number of corners at which the parity flips. This will be the subject of the next section.

6 Turning number parity

In this section we will prove that the turning number of a simple closed cycle is always equal to either 1 or -1.

This result is known already for suitably well-behaved curves in the plane [7] and also for polygons [2], but to keep this paper somewhat self-contained, we will present a proof from first principles.

First, we must define precisely what kinds of directed paths we allow.

Definition 1. *A directed path consists of a sequence of steps of unit length going either up, right, left or down. We furthermore require that whenever a path changes direction, it does so by* $\pm 90^\circ$*. We say a path is* closed *if the last step of the path ends at the beginning of the path. We say a path is* simple *if it does not cross or touch itself, that is, every point is the source or target of at most two segments of the path.*

Definition 2. *A* corner *of a given path is any point at which the path makes a turn. We use the notation* $\ulcorner$ *for a corner at which the path moves up and then right, and define* $\urcorner$, $\llcorner$, $\lrcorner$, $\ulcorner$, $\urcorner$, $\llcorner$, *and* $\lrcorner$ *similarly. In arithmetic expressions, we use the same notation to represent the* number *of such corners in the path in question. Thus,* $\ulcorner + \urcorner$ *represents the total number of* $\ulcorner$ *and* $\urcorner$ *corners in the given path.*

We will first present a way of calculating the *turning number* of a path, i.e. the number clockwise rotations a person following the path would make. Note that this need not be a whole number.

Intuitively, whenever the path turns to the right locally ($\ulcorner$, $\urcorner$, $\llcorner$ or $\lrcorner$), the turning number increases by ¼, and conversely whenever it turns to the left locally ($\ulcorner$, $\urcorner$, $\llcorner$ or $\lrcorner$), the turning number decreases by ¼. Based on this, we define the turning number of a path as follows:

Definition 3. *The* turning number *of a path is defined as*

$$¼(\ulcorner + \urcorner + \llcorner + \lrcorner) - ¼(\ulcorner + \urcorner + \llcorner + \lrcorner).$$

Of course, this definition is a bit cumbersome to work with, but luckily there is an easier way to calculate the turning number in the case of *closed* paths. To see this, we first need the following lemma:

Lemma 1. *For any closed path, the following equalities hold:*

$$\ulcorner - \lrcorner \;=\; \lrcorner - \ulcorner \;=\; \llcorner - \urcorner \;=\; \urcorner - \llcorner$$

Proof. Consider any ↱ or ↳ corner. Following the path in the direction of the arrow, we must eventually end up at a ↥ or ↴ corner, since the path is closed. A similar argument shows that any ↥ or ↴ corner is preceded by a ↱ or ↳ corner. Thus, we have the following equality relating the number of such corners in the path:

$$↱ + ↳ \;=\; ↥ + ↴.$$

By rearranging this equality, we get

$$↱ - ↥ \;=\; ↴ - ↳,$$

which gives us one part of the desired equality. The remaining equalities follow in the same way. □

Corollary 1. *The turning number of a closed path is given by* $↱ - ↥$.

Proof. Rearranging the definition of the turning number, we find that it is equal to

$$¼((↱ - ↥) + (↰ - ↰) + (↵ - ↶) + (↴ - ↳)).$$

By the preceding lemma, each of the four differences is equal to $↱ - ↥$, and hence the entire expression reduces to simply $↱ - ↥$. □

In addition to this, we also have the nice result that the turning number of a closed path is well-behaved with regard to various transformations of this path:

Corollary 2. *Rotating a closed path does not change the turning number. Mirroring or reversing the direction of a closed path inverts the turning number.*

Proof. We show here one of the cases. Consider a given closed path. By the previous corollary, it has turning number equal to $↱ - ↥$. If the direction of the path is reversed, every ↱ corner becomes a ↶ corner, and every ↥ corner becomes a ↵ corner. Thus, the value of $↱ - ↥$ is equal to $↶ - ↵$ after reversing the direction of the path. We now have

$$↶ - ↵ = -(↵ - ↶),$$

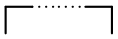
and from the previous lemma, it follows that $↵ - ↶$ is the turning number of the reversed path. We thus conclude that reversing the direction of the path inverts the turning number. The remaining cases are similar. □

Next, we need to show that any closed, *simple* path has a turning number of either $+1$ or -1. We prove this in two steps. First, we show that any closed, simple path can be reduced to a path with only four edges (i.e. a square) by a sequence of local reductions. By observing that the local reductions preserve the turning number, we get the desired result.

Theorem 1. *Any closed, simple path with length greater than 4 can be reduced using local reductions to a closed, simple path with a strictly smaller length and with the same turning number.*

Proof. Formally, we prove this by induction on the length of the path. We will leave these appeals to the induction hypothesis implicit, however, and simply present the reductions. Additionally, we will present the reductions in terms of *undirected* paths, and only consider the turning number once we've established exactly what the reductions are.

First, we want to find an appropriate place to reduce the path. This will be a horizontal segment of some length where each end of the segment points downwards:



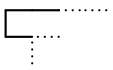
First, however, we must establish that there must exist such a segment. We do this with a sequence of observations:

Observation 1: There is at least one horizontal segment. As the path has length greater than 4, it must contain *some* segment, either horizontal or vertical. If the segment is horizontal, we are done. If the segment is vertical, we can follow it upwards until it turns (which it must, as the path is closed), and there we will find a horizontal segment.

Observation 2: The topmost horizontal segment must turn downwards at each end. If at either end it turns upwards, we may follow the path upwards until it turns again. At this point, we will have found a horizontal segment which is higher up than the topmost horizontal segment, and this is a contradiction.

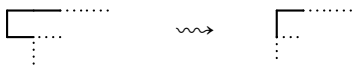
Having now established the existence of the desired segment, we will pick a segment of this form for which the length is *minimal*, i.e. no segment of this form with strictly smaller length exists. It is this segment we will reduce locally.

We first consider the case where the left end of the segment turns *under* the segment. We ignore the right end of the segment for now:

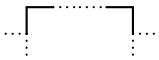


The dotted lines at the bottom indicate the directions in which the path may proceed. Note that the path cannot turn up, as this would make it either non-simple or of length exactly 4.

In this case, we reduce the path as follows:



In a similar fashion, we can reduce the right end of the segment in the cases where it too turns underneath the segment. This leaves the following remaining case:



We would like to reduce it as follows:

To do so, however, we need to argue that this does not make the path intersect itself, otherwise we would be unable to apply the induction hypothesis. We will therefore establish that there cannot be any horizontal segments immediately below the given horizontal segment. Assume for the purposes of contradiction that there exists such a segment. Following the segment to the left, it must eventually turn downwards. Turning upwards would make the path intersect itself. Similarly, the right end of the segment must also turn downwards. At this point, however, we would have a horizontal segment, turning down at both ends, and with a length that is strictly smaller than the length of the segment we started with, and this is a contradiction. As we have now established that there are no horizontal segments directly beneath our chosen segment, we may now reduce it as previously shown.

This takes care of all possible cases. All that remains now is to note that the above reductions preserve the turning number. This can either be done the *hard* way, by checking every single case separately, or it can be done the *clever* way, by using Corollary 2. Note that rotating the path does not change the turning number, and reversing and mirroring it only changes the sign of the turning number, hence it is sufficient to observe that all the configurations seen in the reductions above can be rotated, mirrored or reversed in such a way that there are no $\ulcorner$ or $\urcorner$ corners either before or after the reduction. It is then immediate that all the reductions preserve the turning number. □

Armed with the above theorem, we may now prove that the desired property of closed, simple paths indeed holds.

Corollary 3. *The turning number of a closed, simple path is either* 1 *or* -1.

Proof. Using the previous theorem, any closed, simple path of length greater than 4 may be reduced to one with a strictly smaller length, without changing the turning number. All that remains, then, is to check that all closed paths of length at most 4 have turning number ± 1. There are exactly two of these paths, and the result thus follows from a simple inspection of these. □

Now that we have established that

$$\ulcorner - \urcorner = \pm 1, \qquad \text{and thus} \qquad \ulcorner + \urcorner \equiv 1 \mod 2$$

we have shown that the constraint presented in the previous section indeed has the property that any closed simple cycle induces an odd number of disequalities, and thus leads to a contradiction.

7 From grid graphs to general planar graphs

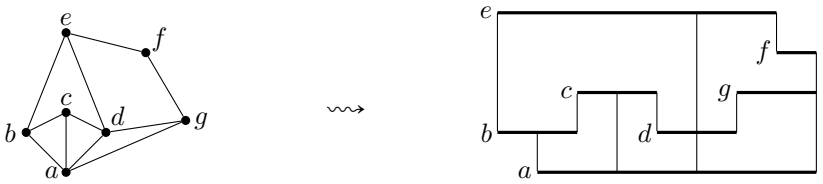
In this section, we will show how to extend the results of the previous section to general planar graphs.

One obvious way of doing this would be to use the fact that any planar graph can obviously be approximated by a suitable subgraph of the grid graph, by suitably "rasterising" an embedding of the planar graph. This would not be

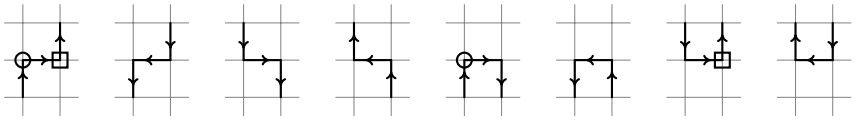
particular efficient, however, as the approximation might use many more vertices than the original graph.

Instead, we will show how a certain kind of planar graph embedding gives rise to a straightforward way of assigning appropriate turning number parity constraints to a general planar graph.

Our main tool for this purpose will be the so-called *visibility representation* [3, 4] of a planar graph. Put briefly, any planar graph can be represented in a form where every edge is a vertical line, and every vertex is a horizontal line:

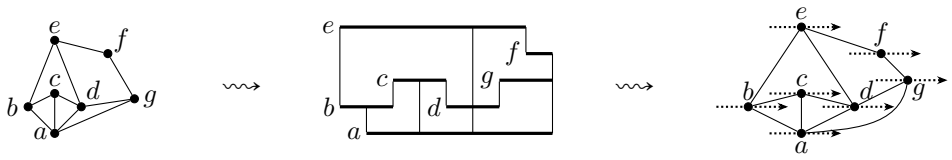


Now, this is already a much better representation than simply approximating an arbitrary planar embedding. First, note that the turning number parity does not change along any vertical edge, hence we can simply think of these as very *tall* single edges. Thus, we can easily represent any planar graph as one that is locally grid-like, by putting d vertices for each vertex of degree d in the original graph. However, we can in fact do better still. To see this, consider the ways in which a path can traverse a vertex in the visibility representation. If the path crosses the vertex without changing direction from up to down or vice versa, it is clear that the turning number parity does not change. This takes care of four of the possibilities. If, on the other hand, the path changes direction at the vertex, we must consider whether it moves leftwards or rightwards across said vertex. In the former case, the parity is unchanged, and in the latter case, it is flipped. This fully sums up the local behaviour of paths going through a vertex.



The key observation now is that we can enforce exactly this behaviour on any straight-line embedding of the planar graph. To do this, we will mark each vertex in the original graph embedding with information from the visibility representation that will enable us make the former act exactly as the latter. We will mark this as a small dotted arrow traversing each vertex. Every edge to the left of the arrow (when looking in the direction the arrow is pointing) will be among the edges above this vertex in the visibility representation, and dually the edges on the right of the arrow will be the edges below the vertex. Similarly, the edges will be ordered according to which order they occur around the vertex, using the direction of the arrow to distinguish between the possibilities. Essentially, one can think of this representation as one in which all the vertices

in the visibility representation have been contracted into a single vertex again.



The question of whether the parity should flip when traversing a vertex in this graph can now be easily explained in terms of this additional arrow: if the path "bounces off" the arrow, and is travelling in the direction the arrow is pointing, the parity should flip. In all other cases, the parity should remain the same.

8 Further graph constraints

In this section, we will consider a variety of further graph constraints that may be achieved by adapting our acyclicity constraint.

First, let us consider the problem of connectedness. Given a subset of the vertices, we would like to enforce that the subgraph containing exactly these vertices (and the edges that connect them with each other) is in fact connected. For the sake of simplicity, we will assume that we have selected one vertex r that is guaranteed to be in the subset. We now enforce the constraint as follows:

- we associate a 2-valued variable $s_{i,j}$ to each vertex, representing whether said vertex is part of the selected subset.
- For each vertex in this subset, except for the vertex r, we require that at exactly one of its edges is selected, and that it points *away* from the vertex:

  $$\forall (i,j) \in \mathbb{Z} \times \mathbb{Z} \backslash \{r\}. \quad (h_{i,j} > 0) + (v_{i,j} > 0) + (h_{i-1,j} < 0) + (v_{i,j-1} < 0) = 1$$

  This constraint subsumes the constraints from Section 3 that forced the solution to contain only simple, directed paths, hence we discard these constraints.

The effect of these requirements is that our selected edges now induce a *tree* structure on the subgraph induced by the selected vertices. The vertex r then acts as the root of this induced tree. As there is only one root vertex, and as the tree will contain only edges from the subgraph, it is immediate that the subgraph in question must be connected.

Again, the acyclicity constraint becomes crucial — without it, a tree may end in a cycle, and thus might fail to be eventually connected to the root. All that remains, then, is to ensure that the possibility of having multiple edges entering a vertex does not break the acyclicity constraint we have already defined. The only way this could happen is if there were some way to constrain a given edge to take on two unequal values at the same time. As we only constrain the value of an edge entering a vertex based on the local configuration of it and the edge (if any) leaving the vertex, and as there is at most one edge leaving a vertex, it is impossible for the acyclicity constraint to overconstrain the variables. Note that this is not the case if we allow more than one edge to exit a given vertex.

8.1 Single-cycle constraints

Previously, we have seen how to constrain the solution to consist of a single path. We will now consider how to constrain the solution to consist of a single cycle instead. Off-hand, this may seem a bit strange, as we would then simultaneously constrain the solution to contain *no* cycles, and yet also a single cycle, which would be a contradiction.

To get the desired behaviour, we will first adapt the acyclicity constraint to allow this constraint to be broken at exactly one vertex in the graph. We first associate a 2-valued variable $b_{i,j}$ to each vertex. This variable will be constrained as follows:

- Among all the vertices, at most one of them can have $b_{i,j} = 1$. This is easily enforced by the following constraint:

$$\sum_{i,j} b_{i,j} = 1$$

- At any vertex, the acyclicity constraint is only enforced if the value of $b_{i,j}$ is equal to zero. For each constraint $C(i,j)$ concerning the vertex (i,j) this amounts to adding a precondition as follows:

$$\forall i, j. \quad (b_{i,j} = 0) \supset C(i,j)$$

At this point we have enough to encode the single-cycle constraint, but in fact we can do even better in this particular situation. One drawback of the acyclicity constraint we presented previously is that it requires a domain of size 3 for the variables associated to edges. If, on the other hand, our solution is known to consist of only cycles, we do not need this extra information to be encoded in the edges. The crucial observation is the following: if the solution consists of only cycles, we can keep track of the number of cycles (modulo 2) that any given face of the graph is inside. This allows us to associate a direction to each edge by stating that the face to the left of a given edge (as seen in the implicit direction of the edge) is always the outside face.

The benefit of this optimisation is clear: instead of constraining the 3-valued edges to ensure that they give rise to directed cycles, we may assume they are 2-valued, and simply add a single 2-valued variable $f_{i,j}$ for each face of the graph. As there are generally more edges than faces in a given planar graph, this is an improvement. If we define that $f_{i,j}$ is the face that is immediately above $h_{i,j}$ and immediately to the right of $v_{i,j}$, it is a straightforward matter to propagate the inside/outside information by the following constraints (recall that the edges now have domain $\{0,1\}$):

$$\forall i, j. \quad f_{i,j-1} = f_{i,j} \oplus h_{i,j}$$
$$\forall i, j. \quad f_{i-1,j} = f_{i,j} \oplus v_{i,j}$$

Where we previously used, say, $h_{i,j} < 0$, we may now instead put $(f_{i,j-1} = 0) \wedge (f_{i,j} = 1)$, and thus restate the constraints from Section 5 in terms of the $f_{i,j}$ variables.

In fact, by doing this, we can eliminate the $h_{i,j}$ and $v_{i,j}$ variables *entirely*, and rewrite any expression referring to these variables into one using the $f_{i,j}$ variables instead.

9 Conclusion and Future Work

In this paper, we presented a novel way of encoding acyclicity constraints for planar graphs by means of a notion of *turning number parity*.

Although this encoding is efficient in terms of space, it is less clear whether it is actually practical to use. One benefit of the basic encoding is that cycles may be detected by simple interval constraint propagation where

$$x \in \{k, \ldots, \ell\}, \quad y \in \{m, \ldots, n\}, \quad x < y$$

immediately induces the constraints $k < y$ and $x < n$, and thus constrains the domain of x and y to be $\{k, \ldots, \min(\ell, n-1)\}$ and $\{\max(k+1, m), \ldots, n\}$ respectively. If moreover we know that $y < x$, propagating the above constraint will eventually exclude all possibilities in the domain of either x or y, and thus create a contradiction.

For the refined constraint, the picture is less clear. Essentially, we induce a sequence of equalities and disequalities between the parity variables in our directed cycle. If we decide on the parity of some variable in a path, the consequences of this choice will immediately propagate to every other vertex in the path, and if the path is in fact a cycle, this propagation will ultimately fail. If, on the other hand, no such choice has been made, the constraint propagation engine may fail to detect the contradiction. Consider for instance the following (dis)equalities:

$$a \neq b, \quad b = c, \quad c = d, \quad d = a, \quad a, b, c, d \in \{0, 1\}$$

With these constraints, there is no local way to make progress by pruning the domains of the variables.

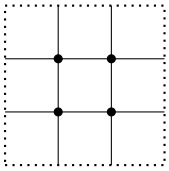
On the other hand, only a small bit of propagation is required to obtain a contradiction. If we add the following propagation rules

$$x \neq y, y = z \Longrightarrow x \neq z \qquad\qquad x \neq y, y \neq z \Longrightarrow x = z,$$

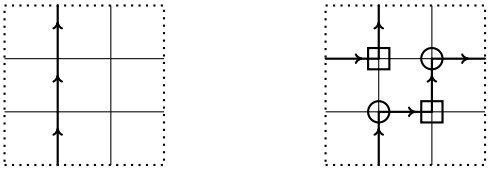
and apply it to the above example, it will eventually derive $a \neq a$, and get a contradiction. In fact, for path and cycle constraints, we can use the above as *simplification* rules, and allow them to consume both of the input constraints (for the connectedness constraint, this is no longer valid).

As we have shown, there exists an efficient encoding for *planar* graphs. It would be interesting to see how one might adapt the present constraints to handle *non-planar* graphs as well. One thing that should be noted is that the present approach does not immediately extend to more general graphs. Consider for instance the following graph embedded on the following planar representation of the torus (where the top and bottom edges are identified, and likewise for the

left and right edges).



Here, it is easy to construct cycles that e.g. have no corners at all — in which case the turning number parity certainly doesn't change — or an even number of corners:



Thus, it is necessary to add further constraints to ensure that these cycles are also excluded.

Acknowledgements This work is funded by the ERC Advanced Grant *ProofCert*.

References

1. M. Gebser, T. Janhunen, and J. Rintanen. Sat modulo graphs: Acyclicity. In *Logics in Artificial Intelligence*, pages 137–151. Springer, 2014.
2. B. Grünbaum and G. C. Shephard. Rotation and winding numbers for planar polygons and curves. *Transactions of the American Mathematical Society*, 322(1):169–187, 1990.
3. R. Otten and J. Van Wijk. Graph representations in interactive layout design. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 914–918, 1978.
4. P. Rosenstiehl and R. E. Tarjan. Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete & Computational Geometry*, 1(1):343–353, 1986.
5. N. Tamura. Solving puzzles with sugar constraint solver. Slides, August 2008. http://bach.istc.kobe-u.ac.jp/sugar/puzzles/sugar-puzzles.pdf.
6. N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear csp into sat. *Constraints*, 14(2):254–272, 2009.
7. H. Whitney. On regular closed curves in the plane. *Compositio Mathematica*, 4:276–284, 1937.